Electronic Version
Stylesheet Version v1.1.1

Description

MULTIPROCESSOR CODE FIX USING A LOCAL CACHE

BACKGROUND OF INVENTION

[0001] 1. Field of the Invention

[0002] The present invention relates generally to a multiprocessor code fix using a local cache, and more particularly pertains to a multiprocessor code fix system and method wherein operating code fixes are supplied to the multiple processors which utilize the same operating code by storing the correction code fixes in a central RAM, and distributing the code fixes over a dedicated code fix bus to a local cache for each processor.

[0003] 2. Discussion of the Prior Art

[0004] As ASICs (Application Specific Integrated Circuits) have become larger and more complex, the use of ASICs in microprocessors and other programmable logic has become more common. The code or software that is used to con-

trol these programmable devices must be stored in memory during functional operation. The memory used for the storage of the code can be either RAM (Random Access Memory), ROM (Read-Only Memory) or NVRAM (Non-Volatile RAM). Each of these memory structures has its advantages and disadvantages for storage of the operation code. RAM memory allows the user to change the code at any time after manufacturing, however the code still needs to come from somewhere at power-up time since it disappears after each powering off. ROM memory is the smallest and requires the least power to run, however since the code storage is created during manufacturing it is not able to be changed afterward. Thus a bug or problem in the code means that the ASIC must be thrown away. NVRAM memory compromises between RAM and ROM and allows the code to be written into it after manufacturing but retains the code even when no power is applied to the circuit. Unfortunately, the size, speed, and manufacturing cost of nonvolatile RAM is not competitive with ROM and RAM.

[0005] Figure 1 illustrates one practical prior art solution to this code storage problem which uses ROM for the majority of storage, and then place breaks in the code between sec-

tions of code that check in a RAM storage area to see if the next section of code is valid. The RAM is loaded at power up from an off chip EEPROM (Electronically Erasable/Programmable ROM). The ROMs inside the ASIC contain jump tables for each section of code. When the ROM reads the first section of code, it is told to jump to a location in the RAM. If the code is OK, the RAM simply sends back a return statement and the code is executed from the ROM.

[0006] If, however, there is a bug in the ROM code, as illustrated at addresses/locations 0x100C and 0x1010 which contain bad code, when the ROM reads the first section of code at address 0x1008, and is told by instruction JUMP 0x0FEC to jump to address 0x0FEC in the RAM. The RAM contains the fixed code at the next addresses 0x0FF0, 0x0FF4, 0x0FF8, and thus the processor runs from the new code in the RAM until the end of the section at address 0x0FFC, where the RAM gives the return command to JUMP 0x01024, and execution resumes from address 0x1018 in the ROM. The drawbacks to this method are that it takes time at each powerup for the RAM to load in the fixed code from the EEPROM.

[0007] The solution of Figure 1 optimizes the use of ROM (the

smallest and least power usage memory), and uses the smallest amount of RAM requiring loading from power-up. This solution also allows for the possibility that no code changes are needed and no RAM loading required. However, it requires that a small amount of processing time is used for checking the validity, and a small amount of storage space is used for the branching code. One problem is determining the granularity of the breaks and the amount of RAM included in the design. This granularity and RAM storage must be predicted before manufacturing and with no knowledge of the extent of the problems present in the code.

[0008] The solution of Figure 1 works well given the limitations of technology, however as the number of processors on a chip increases, as in chips used in internet processors for example, the amount of RAM for fixes increases and the amount of time or buses required to load the RAM at power-up increases. A typical group of four processors (a square works best since it uses minimal space) requires four times the resources for the bug fixes. If four of these processor groups are then further grouped, the resource demand increases by sixteen.

[0009] From a software perspective, there is an additional factor

in the use of more processors. Sixteen processors with independent code require sixteen times the software effort. What is effective in many applications is to duplicate the code in all of the processors and to use these processor groups as processing engines for multiple channels of data. For example, a modem can use multiple processors to handle multiple communication channels. This approach limits the amount of software effort required to take advantage of the use of multiple processors.

[0010] The problem of code fixes now becomes a case of the same fixes being duplicated over each processor. (The use of a single ROM for the processor becomes intractable as the number and speed of processors increases).

SUMMARY OF INVENTION

[0011] The present invention provides a multiprocessor code fix system and method that uses a local cache for each processor wherein operating code fixes are supplied to the multiple processors which utilize the same operating code by storing the correction code fixes in a central RAM, and distributing the code fixes over a dedicated code fix bus to a local cache for each processor.

[0012] The present invention provides the following advantages relative to the prior art. It minimizes the amount of RAM

required to implement a ROM-jump-patch type of ROM fixing scheme. By having the local caches for the processors all connected to the same fix bus, processors that are running the same code fixes in the same time frame have the code fix already in their local cache. The present invention also eliminates the problem of guessing how much RAM space is required for the fix. A full size RAM could be used as the central fix RAM with the small memory caches holding the local information.

[0013] The subject invention also minimizes the amount and time required for loading the fix RAM from external memory.

BRIEF DESCRIPTION OF DRAWINGS

[0014] The foregoing objects and advantages of the present invention for a multiprocessor code fix using a local cache may be more readily understood by one skilled in the art with reference being had to the following detailed description of several embodiments thereof, taken in conjunction with the accompanying drawings wherein like elements are designated by identical reference numerals throughout the several views, and in which:

[0015] Figure 1 illustrates one practical prior art solution for code storage which uses ROM for the majority of code storage and places breaks in the code between sections of code

that check in a RAM storage area to see if the next section of code is valid.

[0016] Figure 2 illustrates an embodiment of the present invention in which each processor of the system is provided with its own ROM and also with a small fix cache which is provided with code fixes distributed from a central RAM.

[0017] Figure 3 illustrates an embodiment of the present invention having a plurality of dedicated fix buses and fix caches to capture a corrected code from a RAM.

[0018] Figure 4 illustrates a code example wherein a fix cache contains corrected code, and code is being executed in the processor from a ROM, and when the jump 0x0FEC opcode is executed, the processor goes to its fix cache and begins executing the new code associated with address 0x0FEC, and when the new code is finished executing, the new code directs the processor to resume execution from the ROM.

[0019] Figure 5 illustrates a code example wherein the fix cache does not contain corrected code.

[0020] Figure 6 illustrates a logic flow diagram for code fixes using multiple caches, a fix bus and a fix RAM, and describes the process by which the fix code is executed by the processor.

DETAILED DESCRIPTION

[0021] The present invention provides a solution to the problem of code fixes in a system having multiple processors utilizing the same code. The present invention solves the problem of code fixes with multiple processors by storing the correction code in a central RAM and then distributing the code fixes as needed into a local cache for each processor and keeping the remainder of the fix caches coherent in time. This coherency relies on the locality of code operation within the other processors. The first processor encountering a code fix incurs an access time penalty of getting the fix from the RAM, but the remainder of the processors have direct access to the code fix from their own caches which are automatically updated with the new code.

[0022] Figure 2 illustrates an embodiment of the present invention in which each processor 10 of the system has its own ROM 12 which stores its operating code, and also has a small fix local memory cache 14 which contains or will soon contain the code fixes distributed over a fix bus 16 from a central RAM 18. The concept of a cache is well known in the art, and is a local memory storage area that keeps frequently accessed data or program instructions

readily available so that the processor does not retrieve them repeatedly from slow storage. The RAM 18 is loaded at power up from an off chip EEPROM (Electronically Erasable/Programmable ROM) 22 over a bus 20. Assuming during a run operation that not all processors are at the exact same location in the code at the same time and also assuming that the code is about 90% correct in the first place, the problem of code fixes is solved in the present invention by using one or more dedicated fix buses 16 and the fix caches 14 associated with individual processors to capture a corrected code from the RAM 18.

[0023] Figure 3 illustrates an embodiment of the present invention having a plurality of dedicated fix buses 16 and fix caches 14 to capture a correct code from a RAM 18.

[0024] Since some RAMs today can have four ports, a system can have four buses 16 on each RAM, as illustrated in Figure 3, or could have a different number of buses. Each fix bus 30 then supports a certain number of fix caches 14, each associated with an individual processor 10. The RAM 18 is connected by a bus 20 to an off-chip EEPROM 22 that simultaneously loads corrected code to the on-chip fix RAM 18 at power-up. Figure 3 illustrates a system having one RAM 18 and multiple fix buses 16 serving a multiple

number of fix caches 14, each having an associated processor 10, with only a few being illustrated in the drawing.

[0025] An embodiment might have a multiple number of RAMs, with each RAM having one or more fix buses which then support a certain number of fix caches. Each RAM would be connected to an off-chip EEPROM that simultaneously loads corrected code to the on-chip fix RAMs at power-up.

[0026] When a processor 10 comes to a place in the code where it needs to perform a jump, it goes to its fix cache 14 to see if the new code has been loaded from the fix bus 16. If no other processor has previously called the code, then it would still reside in the RAM 18 and not in its associated fix cache 14. The processor then makes a request to the jump address in the RAM, and this request is then sent to the fix RAM. The RAM then outputs the new code on the fix bus(es). All of the other attached fix caches would also pull off and store the fix data. For processors which had not yet gotten to that place in the code, it will be readily available in their fix cache when they do need it and save time and resources by not having to fetch the new code directly from RAM.

[0027] An embodiment of the present invention can implement a

least recently used cache algorithm. The fix cache is loaded with new corrected code fixes sequentially every time a new code fix is present on the fix bus, and the cache keeps track of the least recently used code fixes. When there is no longer any room in the fix cache to store additional code fixes, the cache then replaces the least recently used data with a new incoming code fix. A new code fix is fed to each fix cache on the fix bus simultaneously once a processor requests code that is not already resident in its own fix cache.

[0028] The least recently used cache algorithm can be implemented using extra tag bits in a bit field attached to each line in the cache. As a cache line is used, the bit field is changed to represent a recent hit or usage. When a new write into the cache is required, the appropriate bit fields of lines in the cache are checked and the line with the least recently used tag setting is chosen to be replaced and written over. The number of lines that are possible for a given address is determined by the number of divisions of the cache or the number of ways, as the use of ways is well known in cache technology. The number of ways would be application dependent.

[0029] Figure 4 illustrates a code example wherein a fix cache 14

contains corrected code. In the example of Figure 4, code is being executed in the processor 10 from the ROM 12. When the jump 0x0FEC opcode is executed, the processor 10 goes out to its fix cache 14 and begins to execute the code associated with address 0x0FEC. The opcodes in the processor 10 now reflect the new code. When the new code has finished executing, the code resumes from the ROM 12 as seen by the jump 0x1014 opcode located in the fix cache 14.

[0030] Figure 5 illustrates a code example wherein the cache does not contain corrected code.

[0031] In the example of Figure 5, the corrected code does not reside in the fix cache 14. Here the processor 10 requests the new code from location OxOFEC from its fix cache 14. The fix cache does not contain that particular code fix and sends a request on the fix bus 16 to the RAM 18 for that section of code. The RAM 18 then sends the necessary op codes up to and including the jump instruction out onto the fix bus 16. From there, all attached fix caches 14 ... 14n can load the new data into their cache lines where it is available for future use. Fix cache 14 also loads the new code into its cache lines and feeds the data to the processor 10 for execution. When the op code JUMP 0x1014 is

executed, execution flow returns back to the ROM 12.

[0032] Figure 6 illustrates a logic flow diagram for code fixes using multiple local fix caches, at least one fix bus and a fix RAM, and describes the process by which the fix code is executed by the processor. During execution at 60 the processor executes code from the ROM, and examines the code for a jump at 61. If a jump is not encountered at 61 (NO), at 62 the processor continues execution of the ROM code until a jump command is encountered (YES) to a location outside of the ROM. At 63, the processor goes first to its fix CACHE to execute the new code if it exists at that location (YES). If the new code is in the fix cache, at 64 the new code is executed followed by a jump back to ROM execution. If at 63 the new code is not in the fix cache (NO), at 65 the processor goes to the fix RAM and asks for the data at the jump to address. At 66, the fix RAM retrieves the fix code and sends the new code out on the fix bus to the processor, its fix cache and all of the other fix caches on the fix bus. The RAM sends a fixed amount of data (from 1 to n words depending on the architecture and the designer's discretion) onto the fix bus and then waits until it is asked for new data. At 67, the new code is simultaneously loaded into all of the fix caches on the fix

bus. At 68, the processor executes the new code until it reaches a jump instruction pointing to an address in ROM, and then continues to execute code from the ROM, and so on.

[0033] The advantage of sending a small number of words onto the fix bus is that the RAM is free to deliver other lines of code fixes to other processors running different stages of the software. Additionally, the fix caches themselves can be very small which leaves more real estate on the chip available for other uses.

[0034] The present invention provides the following advantages relative to the prior art. It minimizes the amount of RAM required to implement a ROM-jump-patch type of ROM fixing scheme. By having the caches all connected to the same fix bus, processors that are running the same code fixes in the same time frame have the code fix already in their cache. The present invention also eliminates the problem of guessing how much RAM space is required for the fix. A full size RAM could be used as the central fix RAM with the small memory caches holding the local information.

[0035] The subject invention also minimizes the amount and time required for loading the fix RAM from external memory.

[0036] While several embodiments and variations of the present invention for a multiprocessor code fix using local cache are described in detail herein, it should be apparent that the disclosure and teachings of the present invention will suggest many alternative designs to those skilled in the art.